Programmer en Python, une introduction

November 13, 2016

Objectifs

Être capables de développer et de maintenir de façon collaborative une application d'analyse de données biologique Python.

- Introduction au langage Python.
- Principes de programmation.
- Développement collaboratif.
- Application à la bioinformatique.

Prérequis

Aucun prérequis en matière de programmation.

- Être capable de survivre sous Linux en mode ligne de commande.
- Savoir utiliser un éditeur de texte.

À savoir

- Très peu de slides: Python et GIT disposent de beaucoup de ressources (pédagogiques, références) auxquelles nous renverrons autant que possible.
- Basé sur l'exemple et les travaux pratique: il faudra coder, écrire, collaborer.
- Non exhaustif.

Python Origin's

Crée à la fin des années 80 par *Guido Van Rossum pour*, pour faire autre chose, pendant ses vacances de Noël.

- Initié aux Pays-Bas dans le cadre du projet Amobea (mort en 1996)...
- ..puis au CNRI, puis à BeOpen, puis à la Python Software Foundation.

Des raisons du succès

Initié comme langage de script, Python promeut favorise avant tout la facilité et le confort d'utilisation.

- La plupart des librairies "connues" possède une interface Python.
- Il est facile de "bricoler" un script qui vous passe par la tête.
- Le langage offre *une* façon, facile à retenir, de faire les choses.
- La documentation fait très souvent référence au Monty Python's Flying Circus.
- ...

les inconvénients

Dépendent de vos attentes.

- Fait peu de de cas de la compatibilité ascendante → portabilité déplorable.
- Ne se considère pas responsable des performances d'exécution...
  - qui sont rarement critiques,
  - et souvent plus subtiles que prévues.
- Promeut parfois l'idée que développer est une activité facile.

Gestion de version

Si un code n'est pas appelé à évoluer, il ne mérite probablement pas d'être écrit.

- **Versionnage**: Besoin de garder un trace de son travail.
- **Travail collaboratif**: Besoin de travailler ensembles sans se marcher sur les pieds.

Les outils aident (beaucoup), mais ne dispensent pas de méthodologie.

GIT

Un “stupide pisteur de contenu” initialement créé par Linus Torvalds, rendu utilisable par de fidèles disciples.
Peut signifier, selon l’humeur:

- Un ATL prononçable, libre, que l’on peut prononcer incorrectement “get”, ce qui peut avoir un sens. Ou pas.
- En argot anglais, peut signifier odieux, stupide, méprisable, simplet... quelque chose de clairement péjoratif.
- **G**lobal **I**nformation **T**racker, quand tout va bien.
- **G**oddamn **I**diotic **T**ruckload of sh*t, quand tout va moins bien.

“I’m an egotistical bastard, and I name all my projects after myself. First ’Linux’, now ’git’.”

Pourquoi GIT?

Parce que!

- Décentralisé, ce qui peut être déroutant pour les non-initiés.
- Rapide, ce qui peut être utile pour les gros projets.
- Assez complexe, ce qui peut être frustrant pour des configuration simple.
- Réputé ne pas se mettre en travers de la route.
- Peut s'interfacer avec d'autres systèmes.

Autour de GIT

GIT supporte plusieurs modèles de développement, et offre parfois des outils associés.

- Le *modèle dictatorial* (exemple: Linux), modèle pyramidal où chaque membre fait confiance à un nombre restreint de sous-fifres.
- Le modèle collaboratif, à base de “pull requests”, mis en œuvre par GITHub notamment.
- Le modèle “moi et mon laptop”.

L'interprète

(ou *interpréteur* en franglais)
La *commande* **python** permet d'interpréter du *code* **Python**, au même titre que la commande **perl** permet d'exécuter du code **Perl** etc..)

Quelques types de base

Python propose les types de base usuels, même si ce n'est pas un langage de numéricien.

- Un type flottant, un type entier...
- ... définis comme étant le type "naturel" de la machine.
- Supporte les 4 opération de base, les exposant, le parenthésage..

Exercices:

- Évaluer des expressions, mélanger les types.
- Types complexes ?
- Diviser par zéro.
- Afficher 0.1, l'ajouter à 0.2
- Utiliser des variables à la place des littéraux

Chaînes de caractères

- Séquence de caractères entres `'simple quotte'` ou `"double quottes"`
- Peuvent être affichées avec `print`.
- On peut afficher de longues chaînes sur plusieurs ligne, ou mettre des retour à la ligne dans une chaîne.

Exercices:

- Expliquer `machin[1:3]`
- Utiliser les opérateurs + et * avec des chaînes et des entiers.

Listes

La structure de donnée reine de Python.

- `[3, 5, 7, 11, 13, 17]`
- `['banana', 'grapefruit', 'strawberry', 'lemon']`
- `[3, 'banana', 5, 3.14 ]`
- `[[3, 4, 5], ['strawberry', 'lemon']]`

Exercice:

- "Additionner" deux liste
- Extraire des sous listes.
- Remplacer un élément.
- Remplacer une sous-liste.
- Obtenir la longueur d'un liste.

Conditions

Un algorithme doit faire des choix.

```
if terre_plate:
    print "Pericoloso sporgersi."
```

Pour cela on a besoin d'un notion de vrai et de faux représentées en programmation par la notion d'expression booléenne.

- Une expression booléenne peut avoir deux valeur: `True` et `False`.
- Résultat d'une comparaison (`>`, `<`, `==`, `!=`, `>=`, `<=`)
- ... ou d'une conversion (nombres, liste etc..), éventuellement explicite: **bool(**`<expr>`**)**.

Exercice: tester des expressions: entiers, réels, liste...

Premières boucles

```
i=0
while i<10:
    print i
    i = i+1
```

Exercices:

- Écrire les 10 premiers cubes.
- Écrire les premiers éléments de la suite de Fibonacci.

Premier programme

Dans le fichier *fibo.py*:

```
a, b = 0, 1
max=20
while b < max:
    print b
    a, b = b, a+b
```

À exécuter avec:

```
$ python ./fibo1.py
1
1
2
3
5
8
13
$
```

Premier programme autonome

On souhaite exécuter le programme directement:

- Il doit être exécutable.

```
$ ./fibo2.py
bash: ./fibo2.py: Permission denied
$ ls -l ./fibo2.py
-rw-rw-r-- 1 [...] ./fibo2.py
$ chmod a+x ./fibo2.py
$ ls -l ./fibo2.py
-rwxrwxr-x 1 [...] ./fibo2.py
$ ./fibo2.py
./fibo2.py: line 1: [not happy]
$
```

Premier programme autonome

Il faut également préciser comment il doit être exécuté:

```
$ emacs ./fibo2.py
$ more fibo2.py
#!/usr/bin/python
a, b = 0, 1
[...]
$ ./fibo2.py
1
[...]
13
$
```

On préférera la ligne shebang suivante:

```
!#/usr/bin/env python
```

Arguments

On souhaite pouvoir préciser combien de nombre de Fibonacci le programme affichera.
Exercice: Retrouver la documentation concernant l'invocation de l'interprète, un indice s'y trouve peut-être.
Exercice: Modifier le programme pour que l'on puisse lui transmettre ce nombre.

GIT, phase d'approche

Une première utilisation, minimaliste, de **git** consiste à s'en servir de versionnage local.

- On empile les versions, mais sans avoir à gérer une pléthore de copies.
- en cas de problèmes, on peut retrouver n'importe quelle version.

Documentations:

- Sur le web: `https://git-scm.com/doc`
- En ligne de commande: `git help`

Premier dépôt

Exercice: créer un dépôt local pour gérer l'évolution des codes développés pendant la formation.

- `git help init`
- `git help add`
- `git help commit`
- `git help checkout`
- `git help status`

Test, Arguments

Exercices:

- On souhaite rendre l'utilisation de argument obligatoire, et afficher un message explicatif si cet argument unique n'est pas fournis.
- "Commiter" la nouvelle version.
- Retrouver la version précédente.

Test, Arguments

Exercices:

- On souhaite rendre l'utilisation de argument obligatoire, et afficher un message explicatif si cet argument unique n'est pas fournis.
- "Commiter" la nouvelle version.
- Retrouver la version précédente.

Boucles, liste

Exercices:

- Écrire un programme qui affiche la moyenne de ses argument.
- Lui faire également afficher la moyenne quadratique de ses arguments.

Fonctions

Syntaxe:

```
def moyenne(a,b):
    m = (a+b)/2
    return m
```

Exercices:

- Écrire une fonction calculant la moyenne d'un nombre arbitraire de nombres.
- Reprendre l'exemple précédant en utilisant cette fonction.

Fonctions

La fonction peut être vue comme unité de compréhension. Découper son programme en fonction permet de le diviser en unités compréhensible quand sa taille augmente.

- De l'extérieur: on doit pouvoir comprendre ce que fait la fonction sans avoir besoin de lire son code.
- De l'intérieur: on doit pouvoir vérifier que la fonction fait ce qu'elle est supposée faire.

- Combien de mots vous faut il pour expliquer ce que fait votre fonction ?
- Est-ce que votre fonction fait une chose, et une seule.
- Son nom signifie t-il quelque chose de clair pour les personnes qui seront appelés à l'utiliser.
- Avez vous besoin de dire *comment* elle le fait.

De l'intérieur:

- Un collègue peut-il vérifier que la fonction est correcte (fait ce qu'elle prétend faire, et rien d'autre).
- Sa taille lui permet elle de tenir sur une fenêtre de taille moyenne ?
- Les noms des variables utilisés sont-ils significatifs.
- Est ce que je la comprendrai dans 6 mois.

Récurissivîté

Une fonction peut s'appeler elle même:

```
def fac(n):
    if (n == 1):
        return 1
    else:
        return n*fac(n-1)
```

Saisie utilisateur

La fonction `raw_input` permet de lire ce qui est tapé au clavier.
Exercice:

- Écrire une fonction qui retourne `True` ou `False` suivant que l'utilisateur est entré `oui` ou `non`.
- Autoriser optionnellement les majuscules
- Accepter deux listes: les chaînes correspondant à vrai, et celle correspondant à faux.
- Accepter un paramètre spécifiant le nombre d'essais autorisés.

Paramètres par défaut

Exercice: Donner des valeurs par défaut 'raisonnables' pour tout les paramètres sauf le prompt.

Fonction comme variable

Une fonction est un objet (à peu de choses près) comme un autre.

```
>>> def square(x):   return x**2
 ...
>>> f=square
>>> f(2)
4
>>> g=lambda x: x**2
>>> g(2)
4
>>> f(2) == g(2)
True
>>>
```

Fonction comme argument

Exercices:

- Écrire une fonction **appliquer** prenant en argument une liste et une fonction et affichant les images des éléments.
- Écrire une fonction **image** prenant en argument une liste et une fonction et *retournant* la liste des images.
- Écrire une fonction **combiner** prenant en argument deux fonctions à un argument et retournant quelque chose de sensé.

Modules

Constats:

- Devoir retaper son code à chaque nouvelle session python est fastidieux.
- Devoir écrire un script sur un seul fichier également, mais ce n'est pas le plus grave.
  - Cela implique d'avoir plusieurs copies des fonctions/code.
  - Tout code doit être maintenu (ou est probablement bon à jeter).
    doublerightarrow N copie $\equiv$ N tâche de maintenance

Modules

fibo.py:

```
def fac(n):
    if (n==1): return n
    else: return n*fac(n-1)
```

list_util.py:

```
def appliquer(l, f):
    return [ f(x) for x
```

```
>>> import fac
>>> import list_util
>>> list_util.appliquer(range(1,11), fac.fac)
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 36288
>>>
```

Nommage

Le nommage des entités (fonctions, variables, modules, classe..) est l'un des problèmes les plus sous-estimés par les aspirants développeurs.
Un "bon" nom:

- Un nom doit être significatif pour le lecteur (qui doit être identifié).
- Un nom doit être significatif dans le(s) cadre(s) dans lequel il apparaît.
- Pas trop verbeux pour ne pas ennuyer le lecteur.
- Peut s'appuyer sur des abréviations *si elles sont largement acceptées par la communauté concernée*.
- Peut s'appuyer sur des conventions.

Apprendre à utiliser son éditeur de code est un investissement payant.
Le temps passé pour trouver un bon nom n'est pas du temps perdu.
Il faut essayer d'anticiper les usage possible et/ou probables de son code.

Import

L'utilisation de modules permet de faciliter le travail de nommage.

- En aidant à identifier le contexte d'utilisation. Un nom long étant remplacé par plusieurs nom courts.
- En évitant les conflits sur les noms les plus courants.
- En permettant de renommer une entité lors d'un changement de contexte.

```
>>> from fac import fac
>>> from list_util import appliquer as map
>>> map(range(1,11), fac)
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 36288
>>>
```

```
>>> from fac import *
>>> from list_util import appliquer
>>> appliquer(range(1,11), fac)
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 36288
>>>
```

Modules ou commande ?

Parfois, la limite entre module et commande n'est pas claire: **facto.py** peut être utilisé à la fois comme module et comme commande:
Il est possible de se baser sur le *nom courant* du module pour distinguer un appel de commande d'une importation de module.

```
>>> import fib
>>> print fib.__name__
fib
>>> print __name__
__main__
>>>
```

En terme de bonne pratique, il est possible d'utiliser cette fonctionnalité pour effectuer des tests basiques.

Tuples

- Un tuple est un “paquet” ordonné et non modifiable de valeurs (éventuellement, d'autres tuples).

```
t = 1, 42, 'bob', 3.14
```

- Se consulte comme une liste:

```
nom = t[2]
```

- Peut être utilisé pour affecter plusieurs variables:

```
nom, age = 'Bob', 42
```

- Peut être utilisé pour affecter plusieurs variables:

```
quotient, reste = divmod(42, 7)
```

- Peut être mis dans dans des listes:

```
student=[('Bob', 15), ('Alice', 16), ('Jane',15) ]
```

- Déjà utilisés dans la fonction de fibonacci.

Fichier

Les fichiers peuvent se diviser en:

- fichiers textes
  - Codé en ASCII: un chiffre $\equiv$ une lettre.
  - Interprétables sans outils particuliers.
  - Peut efficace pour stocker des données.
  - Habituellement organisé en lignes.
- fichiers binaires
  - Contient des données brutes.
  - A généralement besoins d'outils dédier et/ou d'une description formelle.
  - Compact et précis.
  - Parfois dépendant de la plate-forme.

fichiers textes en Python

Exercices:

- Trouver la section consacrées au lecture/écriture de fichiers texte dans le tutoriel python.
- Sachant qu'un fichier texte est vu comme une liste de ligne, écrire une programme réaffichant les lignes d'un fichier dont le nom est fourni en argument.
  - Trouver le type d'une ligne (ou pourra s'aider de la fonction `type`, de l'interprète **python**, des fonctions `readline`, et `open`).
  - Chercher le manuel de référence de la librairie standard standard python. Y retrouver la documentation du type **str**.
  - Modifier le programme pour supprimer le retour chariot.
  - Modifier le programme pour afficher le numéro de ligne lors de l'affichage (indice: fonction `enumerate`).

Git et les branches

- Git permet non seulement de versionner, mais aussi de collaborer, y compris avec soi.
- Votre dépôt peut héberger autant de *versions* différentes de votre code que vous le souhaitez.
- Les branche vont nous permettre de:
  - Faire des essais sans compromettre votre code.
  - De travailler sur différentes fonctionnalités en parallèle.
  - De contrôler les intégrations de ces fonctionnalités dans votre code “principal”.

Git et les branches

Exercices:

- Créer un nouveau dépôt avec `git init`
- Y ajouter le script python **facto.py**, l'intégrer, commiter, taper `git branch -v`
- Créer une nouvelle branche nommée **develop**, qu'observe t'on ?
- Ajouter un fichier **README**, contrôle par git, et modifier le script.
- Observer les différences entre les branches **develop** et **master**.
- Sur la branche **master**, modifier, différemment, le script **facto.py**.
- Intégrer les modification de la branche **develop** dans la branche **master**.
- Intégrer les modification de la branche **master** dans la branche **develop**.

Git et remote

Git est (aussi) un système de versionnage *distribué*.

- De même qu'il est possible de "synchroniser" des branches entre elles, il est possible de synchroniser des dépôts.
- Même si cela n'a rien d'obligatoire, on décide souvent de donner à l'un de ses dépôt un rôle de dépôt de référence. Les dépôt sont généralement organisés en arbres de profondeur 1 dont le dépôt de référence est la racine.
- Sans discipline, la situation devient rapidement ingérable.
- Plusieurs plateformes proposent des service d'hébergement de dépôt git: `https://github.com`, `https://about.gitlab.com`...

Exercice:

- Créez vous un compte sur `https://www.github.com`
- Créer un projet **test**.
- Il va falloir indiquer à github quels sont les comptes (couple machine/login) qui vous sont associés. Cela peut ce faire par *échange de clefs ssh*.
  - Trouver la documentation associée sur la page d'accueils.
  - Quelles sont les précautions à prendre avant de générer un couple de clefs?
  - Mettre en œuvre et tester.

Git et remote

Exercices:

- Cloner le dépôt de la formation `https://github.com/bdartigues/python_cnrs_formation`.
- Utiliser la commande `git remote -v` et assurez vous que vous êtes bien sur la branche **master**.
- Dans le fichier `intro/examples/users/users.txt`, ajouter une ligne en respectant le format utilisé.
- S'interroger sur les causes de dysfonctionnement.

GitHub et les Fork

Vous pouvez dupliquer n'importe quel projet public sur votre espace GitHub.
Cela vous permet de travailler sur vos propres modifications sans impacter le projet "original".
Exercice:

- Faire un fork de `https://github.com/bdartigues/python_cnrs_formation`
- Que dit la commande `git remote -v`
- Le cloner sur son espace et répéter les modifications.
- Utiliser la commande `git push`.
- Constater la modification via l'interface web de GitHub.
- Ajouter un remote **main** correspondant à `https://github.com/bdartigues/python_cnrs_formation`.

GitHub et Pull Request

Sans accès “collaborateur” au projet original, vos contributions se font via des *Pull Request* : vous aller proposer votre modification au projet principal, qui pourra l'accepter, ou pas, ou discuter avec vous de sa pertinence.
Exercice:

- Proposez, via une pull request, votre modification au projet original via l'interface web de GitHub.

Collaborateur GitHub

Le modèle Pull Request peut poser un problème de scalabilité lorsque le nombre d'intervenant est grand: Un seul "propriétaire" doit gérer toutes les évaluations de Pull Request.
Ajouter un collaborateur permet de répartir la charge et d'être plus réactif.
Exercice:

- Par binôme, choisir un dépôt principal, et ajouter votre binôme en collaborateur.

Mode de développement

Un modèle de développement courant sur un projet public GitHub:

- Sur le dépôt principal:
  - Un “noyau dur” de développeurs de confiance.
  - Deux branches principales:
    - **master** branche dont l'on tire les release, très surveillée et modifiée avec précaution.
    - **develop** branche courante de développement, sur laquelle tout le monde se synchronize régulièrement. On y commite les “petites” modification ou les travaux finalisés. Mergée sur **master** après vérifications soignées.
    - plusieurs **branches de travail** ou l'on développe une fonctionnalité sans perturber **develop**, que l'on merge régulièrement.
- Par ailleurs, sur les dépôts forkés, n'importe qui pourra contribuer via des pull request.

Aliasing

En Python, il existe deux type de type: les non modifiables (entier, chaînes, tuples) et les modifiable (listes, dictionnaires, et à peut prêt tout le reste.
Exercice:

- Écrire un fonction modifiant l'entier qui lui est passé en argument et vérifier sa valeur une fois sa fonction exécutée.
- Essayer avec d'autre types.
- Avec des tuples.

Tri

Exercices:

- Écrire un programme qui lise le fichier **users.txt** et affiche les utilisateur listés avec le prénom avant le nom de famille. Passer par des tuples peut aider.
- Modifier le programme pour que chaque utilisateur soit préfixé de son numéro de ligne.

Classes

A partir du chapitre sur les classes du tutoriel, déterminer à quoi ressemblerait une classe **Person**.

- Sachant que l'on s'intéresse à ses noms, prénoms, dates de naissance et description.
- Que l'on veut pouvoir initialiser une instance de **Personne** avec ses 4 informations.

Exercice:

- Aller regarder dans la documentation de la librairie standard de Python, plus précisément dans le chapitre **Generic Operating System Services**. Nous cherchons quelque chose pour lire des arguments de la ligne de commande.
- Modifier le programme précédent pour qu'il prennent deux argument, **-i** et **-o** permettant de préciser respectivement le fichier d'entrée et le fichier de sortie.

Tri

Exercice:

- Pour chaque binôme, choisir un dépôt. S'assure de l'existence d'une branche **develop**.
- L'un va afficher la liste triée par nom, l'autre par date, en conditionnant cette action par une option **–byname** ou **–byage** respectivement. Le travail sera effectué sur une branche spécifique à créer.
- Répercuter les changement sur **develop**.
- détruire les branches.

Exceptions